Meeting - A simple audio teleconferencing tool

Jack Jansen

(DRAFT third version of 29-Sep-92)

1. Introduction

Meeting is a simple audio teleconferencing tool. It copies audio input from your microphone to the speakers of all other people participating in the meeting.

The idea of this paper is to provide both a description of how meeting works and a list of things I learned along the way. First the general structure of the program is explained and then the various parts. Each section not only describes that part but also algorithms tried and rejected, possible extensions, etc.

2. Program overview

For a full description of how meeting is used I refer to the manual page, *meeting*(1). The basic idea is that someone, the initiator, starts a meeting on his workstation. Others can then join that meeting. When someone says something the voice data is transmitted to all other parties and played through their speakers. All participants have a window showing faces, icons or names of all other people in the meeting. The window also contains a per-user 'talk' indicator, which lights up when someone is saying something. You can 'address' someone using the mouse, and everyone sees an arrow from you to the person you are addressing. It is possible to select an area of the screen that is grabbed and displayed on the screens of other participants. Finally, if you have video hardware you can have your face replaced by live video.

3. General structure

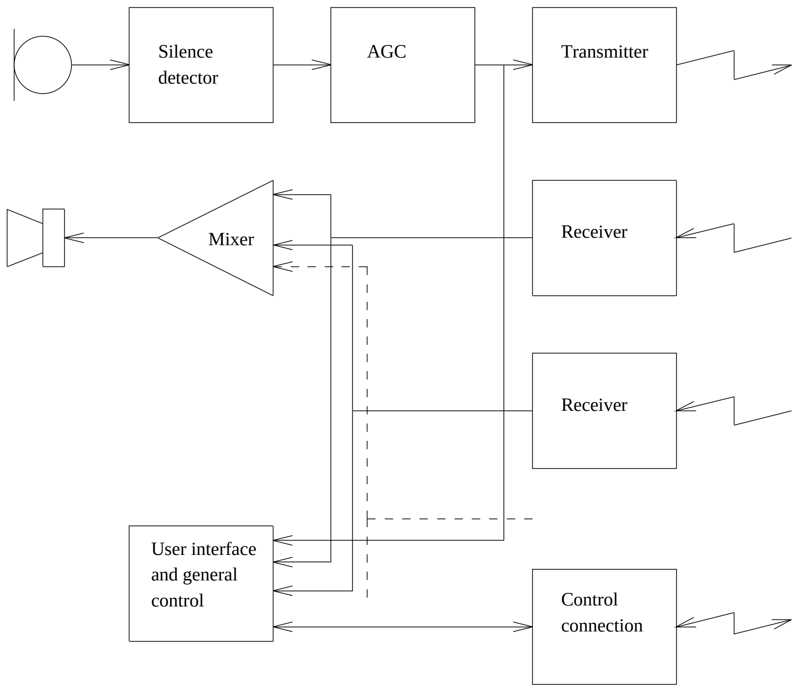
Meeting is written in Python [ref: EurOpen], an interpreted object-oriented programming language developed at CWI. This has a lot of advantages and a few drawbacks. The main drawback is that Python will always be slower than C. One of the effects of this is that the Sun version of meeting does not work nearly as well as the SGI version (when comparing SparcStation 1 or 1+ to SGI Indigoes): the Suns are just too slow.

The advantages of Python are manifold. One is that the whole project took about 10 persondays to complete (up to now, that is:-), and that the first test version was already running after a few hours of programming. An advantage that is derived from this is that Python leads to experimenting. Lots of algorithms have gone through a few generations: it is easy to replace them so there is no severe time-penalty on trying something else. Python programs are also small: the code for the SGI version of meeting is about 1500 lines and added support for the Suns is only 350 lines.

Another important feature of Python is the power of the language with respect to inheritance, modularity and abstract data typing. After the SGI version was finished the Sun version took only about 2 days extra, even though both the display code and the audio code (including a driver in C) had to be written from scratch.

From a signal processing or data-flow point of view the structure of meeting is shown in the figure below.

The code of meeting is structured as a main program, *meeting.py*, containing the main loop and the semantics of the user interface. This main program imports the network code, the user interface module and the audio handling module. In the last two cases, it tries to import the 'classy' SGI interface first, and if that fails it imports the 'no-frills' user interface or the sun audio module. So, in principle a machine with SGI-compatible audio but X-windows display capability only will be supported without modification. The mechanism is such that is easy to try more choices for user interface or audio. There are also two flavors of



networking code, one using Stanford IP multicast and one using normal point-to-point communication. This choice between these two is static, though.

The user interface classes, *window_gl.py* and *window_x.py,* do not share code but have the same interface.

The audio classes *audio_sgi.py* and *audio_sun.py* share a bit of code, the silence detection and automatic gain control, by inheriting the base class *audio_silence.py*.

In the course of the project three C modules have been written that are dynamically loaded into the running Python interpreter. These modules should have a wider application than the meeting program (when they are cleaned up and documented), but they are described here for completeness' sake.

Audioopmodule.c handles audio fragments and can do operations like removing a bias, computing the average or absolute maximum and converting linear to ulaw encoding and vice versa.

Sunaudiodevmodule.c defines a class that allows you to read and write samples to the sun audio device.

Imgfilemodule.c handles SGI imglib images: reading, scaling, etc.

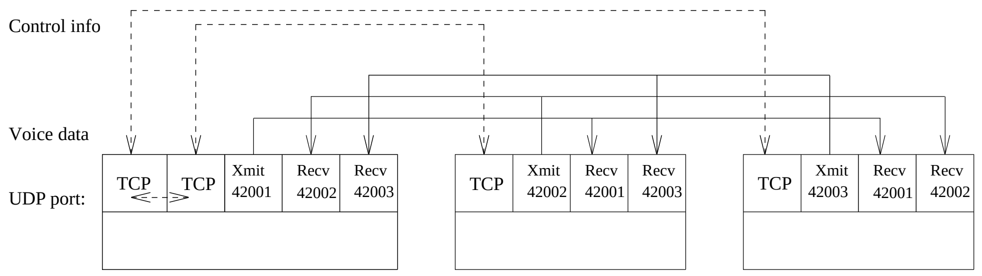
4. Network and protocol handling.

Meeting uses UDP for voice data and TCP for connection management. An earlier version used only UDP, but it was found that loosing connectivity information (due to packet loss or host crash) was bothersome. When some people in the meeting could not be seen by some others communication got very messy. UDP is still used for voice data and some other non-critical tasks, where packet loss is not such a big problem. **[XXXX Measure packet loss]**.

Each instantiation of meeting has a TCP connection to the initiator, a UDP socket on which to transmit voice and, *for each remote machine,* an input socket on which voice packets from other machines are received. The socket structure is sketched in the next figure below.

If meeting is started in initiator mode, it listens for TCP connects on a fixed port. When people join the initiator picks a free port and sends the new member this port and a list of all other people in the meeting (their usernames, hostnames and portnumbers). The new member then opens all relevant sockets, initializes the user interface and sends out a control message (over the TCP connection) to all members to tell that it has joined the meeting. The initiator retransmits all incoming TCP messages over all its other TCP connections.

Whenever a member has voice data to transmit it addresses it to *its own portnumber* on all member hosts.



The multicast network module does this by sending a single multicast message, the unicast module by enumerating over all the member hosts. This scheme has a few advantages over the more obvious scheme of addressing the messages to the portnumber corresponding to the host you are sending to (besides the fact that it is the only reasonable implementation for the multicast code), which was tried in a previous version (and which needed only a single socket in each instantiation of meeting). The main advantage is that the kernel demultiplexes all the UDP ‘streams’ (which works because the kernel does a little buffering of UDP packets). This greatly simplifies the voice multiplexing code described below.

Video is also transmitted in the same way and using the same ports. This might not be such a good solution, since you might well want to use different policies for audio and video.

When a user leaves the meeting the TCP connection is simply closed. The initiator detects this and sends a TCP message to all other participants. When the initiator closes the meeting it closes all TCP connections and all parties detect this and interpret is as the end of the meeting.

The control and data packets are distinguished by their first byte, which indicates the type of the message: ‘D’ for data, ‘Q’ for signoff, etc. Signon and signoff messages are sent over the control connection but some control messages like the ‘gaze’ message (telling that the sender is ‘addressing’ someone, and who) go through the normal data stream. All data (except voice data) is in ASCII. Actually, the data transmitted are the normal ASCII representation of Python data structures, so parsing is trivial. The drawback, however, is that illegal data will undoubtedly crash the program. Messages that go over the control connection are preceded by their length, because TCP does not preserve message boundaries, and the message boundaries are needed to be able to parse the messages reasonably.

An important issue is the optimal packet size. Small packets have the advantage that they allow for shorter latency, but they also incur a bigger overhead per sample (since most of the overhead is per-packet, not per-sample). There is also the maximum packet size supported by the underlying network layers to take into account. UDP supports packets up to 64K in size, but these get fragmented when transferred over ethernet, which supports only packets up to 1500 bytes. Also, experiments have shown that fragmented multicast messages are likely to get lost when traversing multiple networks.

We have picked a packet size of 680 samples. The rationale behind this is that the latency is low enough (85 ms), and that it should always fit in a single ethernet packet: 680 samples is 1360 bytes, leaving 140 bytes for lower layer headers before the packet becomes too big for ethernet. No experiments have been done yet to see whether bigger or smaller packets work better.

Data packets have a sequence number to aid detection of lost packets, duplicates and out-of-order packets. Each instantiation of the listener (there is an instantiation of the listener for each other member host) has a small incoming packet queue. Whenever the ‘*readpkt()*’ method is called it reads all available packets from the socket and sorts them into the queue. It then discards all late packets (packets with a sequence number less or equal than the sequence number of the last packet returned from readpkt) and throws away the first few packets if the queue has become too long (because a long queue means long latency, see the section on audio handling for details). If the sequence number received differs wildly from the one expected all packets are dropped and the protocol code resynchronises. If, after all these actions, the queue is empty an empty packet is returned, otherwise the first packet in the queue is returned.

This relatively complicated handling of incoming packets is the result of lots of experimentation. It was discovered that packets *do* get lost, and *do* arrive out of order. A previous, more simple-minded algorithm that did not try to insert packets into the queue but just dropped them when they were out of order did not perform nearly as well. The current algorithm tries to keep the queue length between two fixed values, 2 and 5 currently, which seems to lead to both reasonable maximum latency and minimum ‘stutter’ because of packets arriving a little late.

A final feature of the networking code is the possibility to code audio data in 4 bit Intel-DVI ADPCM format. This results in packets of only 340 bytes (excluding header) in stead of packets of 1360 bytes. The audio quality is as good not as that of non-compressed audio, however. A previous ADPCM coder, which used 3-bit codes, did exhibit a significant loss in quality so was replaced by this 4-bit coder. Hopefully ADPCM encoding will allow us to hold wide-area meetings (since the used bandwith is only 32Kbit in stead of 128Kbit) but this has not been tested yet. Experiments have showed that a huffman coder after the ADPCM coder would reduce the average bitlength of a sample to 3.2 to 3.5 bits, so it would reduce the bandwidth requirements with another 15%. Finally, we could try using one of the more modern coders like CELP or LPC, which are rumoured to transmit intellegible speech using as little as 5Kbits/second bandwidth.

4.1. Problems and possible extensions.

Protocol freaks will have noticed that the subscription protocol has a race condition when two people join at approximately the same time. Since subscription is a two-phase protocol (first the connection is made and a socket number obtained and then the new member advertises its existence to all others) there is a window in which a member is partially subscribed. This could be fixed by not allowing new subscriptions while another one is still in progress.

There is a cheap extension through which encryption could be done, but I am not sure how strong the encryption is. Given an integer function *F(K,S)* we could send *K*, the key, to all members in the initial signon message. This initial message would have to be encrypted using a standard encryption method, probably some public key encryption scheme. For each message (with sequence number *S* and length *L*) we would then swap the first *F(K,S) mod L* bytes and the last *L-(F(K,S) mod L)* bytes. The resulting message is easily decoded by the receiving side but probably totally unintelligible by outsiders. ADPCM coding strengthens the encryption, and the method could also be made stronger by using multiple cutpoints. As I said, though, I am not sure how easy it is to find *F(K,S)*, maybe using FFTs or something.

5. Audio handling.

One of the main issues in a program of this sort is how to make sure that the timing of the audio data streams is within reasonable bounds. The latency should be low, because delays of over, say, one second seriously hinder conversation. It should not be too low, however, because there are lots of variable delays due to the network, scheduling, etc and striving for minimal delay will inevitably cause stutter in the audio output, which is also very bothersome. A related problem is that of clocks that run at slightly different speeds: this will either lead to frequent stutter in case of a slow transmitter or to long delays in case of a fast transmitter. The last issue involved with timing, that of packet size, has been covered in the previous chapter.

The latency issue is taken into account through the whole program by attempting to keep queuelengths within specified minimum and maximum bounds whenever sound data is being queued. We attempt to keep approximately 0.1-0.3 seconds of sound in the network input queue and at most 0.3 seconds in both the audio input and output queues. There is little use in trying to maintain a minimum audio input queuelength since audio reads are blocking and maintaining a minimum output queuelength by inserting fillers whenever the queuelength was too low has been tried but did not help understandability.

The timing issue has been solved by using the timing of the audio input device as a master clock. Data is read from the microphone in pieces of one packet with a blocking call. This packet is then processed and optionally transmitted. After that at most one packet is taken from each network input queue, all these packets combined and played through the speaker. The minimum network queue length above is not actively maintained by the program but happens more-or-less automatically: whenever a random delay in the network causes two packets from one source to arrive in one ‘clock tick’ the second will remain in the input queue and help to maintain continuous flow of output data.

A subject into which quite a bit of work has been put is that of audio processing on the input side. Silence suppression is definitely needed for an application like this, to cut down on noise levels and network bandwidth. The first simple minded algorithm used a fixed threshold to decide whether a packet read from the microphone contained speech or not: whenever the maximum of the amplitudes was over the threshold the packet was transmitted. This did not work very well: the optimal value of the threshold differed per machine and tended to vary over time (making continuous twiddling of the input volume by the user necessary). It was then discovered that the SGI machines return sound with a DC bias added to it, and this bias is different on every machine. This bias was measured by reading a single packet with the input gain set to zero and calculating the average amplitude over that packet. This bias was then subtracted from all data read subsequently. Incidentally, the Suns have no such bias, either because it is removed in hardware or in the kernel or because the use of U-LAW coding makes the bias disappear.

With this debiasing silence suppression worked better, but still not good enough. After some questions on the net and some browsing I came across [ref], which describes a silence suppression algorithm that is based on both amplitude and number of zero crossings, and is adaptive over time. An initial experiment showed that the number of zero crossings provided no information whatsoever in our case, contrary to what the article said. This might be due to the noisiness of our signals or due to the fact that we work with fragments of approximately 800 samples and the authors of the paper with fragments of 80 samples, I am not sure. The rest of the article was applicable, though, and led to the current silence suppression algorithm.

The algorithm works with a variable threshold and uses a feedback loop to update this threshold. Actually, there are two thresholds: a silence-to-speech threshold and a speech-to-silence threshold, which is 75% lower. Also, if the previous packets were decided to be speech the next 3 packets that fail the (low) threshold are considered to be speech anyway. These two measures manage to keep the microphone open during inter-word gaps and most inter-sentence gaps. This is a good thing, as it improves understandability.

The feedback loop works by maintaining a running average of the average amplitude in all packets that are decided to be silent. By initializing the threshold to some high value the algorithm will adapt itself to the noise level in the room in about 1-2 seconds: initially every packet will be judged silent, and this will bring the threshold down very fast. This feedback loop does not react to ‘bad news’, however: when the level of the background noise rises all packets will exceed the threshold and be judged speech. Therefore, they will not update the threshold and this situation continues to exist. To handle this problem the threshold is also incremented by one for each packet. This will lead to a single speech packet being dropped now and again, but this does not seem to be a problem (probably because the packet being dropped will usually be an inter-sentence-gap packet).

It was noted early in the project that an automatic gain control would also be very handy, as the signal levels are fairly low and people tend to move their head around while speaking, etc. An initial experiment with a combined AGC/silence detector gave disappointing results, so a separate AGC was added. The AGC works per-packet and is fairly standard: it tries to keep amplitudes between 8% and 50% of the dynamic range, and increases or decreases amplification in steps of 10% whenever the signal is outside this range. This amplification is also clamped whenever the signal threatens to go over 90% of the dynamic range by immediately lowering it. This forestalls audio feedback, and shouting matches besides:-).

5.1. Audio - Echo suppression

After using meeting for some time it was noted that some form of echo suppression was needed. Especially if you ran meeting with headphones on and others ran with a speaker you would continually hear everything you said back about a second later. This is rather bothersome.

Some experiments were done on echo cancellation: we send a sweeping tone to the speaker and wait until it comes back over the microphone. We then calculate the delay (in audio-clock ticks) and the attenuation. We can use these numbers to try and cancel our outgoing speech from what comes back in over the microphone. Although not entirely unsatisfactory (the echo cancellation is very good for another application, the computer telephone interface) the results were not good enough for meeting. Due to room reflections the signal was rather fuzzy when it returned, and it is not possible to remove it. A better echo cancellator (probably using filterbanks and measuring room acoustics per audio band) might work but has not been tried due to the amount of work needed.

Another try was to shut off the microphone when the speaker was active. This did not work, since there is no way to break in to a conversation when one of the other parties is talking continuously. The next algorithm was the shut off the speaker when the microphone was active (this is more-or-less what the phone companies do on transatlantic links). This was not satisfactory either, since coughs and gusts of wind caused you to miss half sentences.

The current scheme to forestall echo is a supression approach consisting of two measures. First, whenever the speaker is active we increase the treshold by a factor 4. The effect of this is that a burst of sound from the speaker is not as likely to trigger the silence suppressor, which would cause the feedback. The second measure is to lower the output volume when we are transmitting audio. Experiments seem to indicate that these two measures are good enough to suppress echo. An occasional single packet will get back to the speaker, but this is not too much of a problem. Lowering the output volume is also a lot better than completely muting it.

5.2. Audio - odds and ends

The silence suppressor usually works quite well, but it will occasionally not transmit the first sylabbe of a sentence. This could be fixed by remembering a packet when it is decided to be silent, and transmitting that packet after all if the next packet is judged speech. This would add a little extra latency but it would help the network queuelength control. See the previous section for the tradeoffs involved.

The authors of [ref] state that instead of discarding silence it is a better idea to replace it by prerecorded 'background noise'. It could be that this is because of their application (people listening to prerecorded and preprocessed pieces of text through a headphone), but comments from users (especially users with headphones) seem to indicate that it would apply to us as well. We could transmit a packet of background noise to all parties during startup, to be used as our signal whenever we are silent. Some processing on the background noise will be necessary, though, otherwise you will hear annoying clicks 12 times per second.

6. User Interface.

The user interface is very simple, and does a few things to try and overcome the lack of visual contact. Note that some of the things said in this paragraph are not true for the stdwin-based user interface, which has less functionality than the SGI interface. The differences will be explained later in this section.

The user is presented with a display showing the faces of all other participants in the meeting. Under each face is a '*talk indicator*' that lights up whenever that person is talking. You can do the equivalent of directly addressing someone in a real meeting by pressing the mouse over that persons face. This will result in everyone in the meeting seeing an arrow originating somewhere at your forehead and ending at the chin of the person you are looking at. No serious tests have been done yet, but this paradigm appears to work fairly reasonably. Initially it was enough to move the mouse over someone to address them, but this caused lots of arrows to appear by accident. Specific addressing seems to be uncommon enough that it is no problem that you have to press the mouse.

On the SGI machines, the window is automatically resized whenever people join or leave the meeting. The user can also resize the window, and this will result in all images being scaled to fit the new window (so you can decide how much screen space to use for the meeting window).

Things like lighting the indicator whenever sound data is detected from a certain person, detecting 'address' events and displaying them, etc. are all done through methods of the various objects that are inherited or assigned in higher level objects, again showing the strength of the Python object model.

The sun interface is a lot less nifty, by contrast. It is programmed using stdwin [ref: CWI report CS-R8817] (so this should make it portable to things like Ataris or Macintoshes as well, unlike the audio interface). Because stdwin currently has no easy way of importing pictures in one form or another the faces are replaced by fixed smileys, with the persons name affixed under it. Another difference is that the auto-resize-on-join does not work: the users have to resize the window themselves. A final difference is that you *do* have to click on someones' face to address them.

6.1. Control panel

To be written.

6.2. Work to be done on the user interface

There are a few minor improvements that could be made, like using some sort of real images in the stdwin version (this is being worked on).

The current program places the people in the window in a more-or-less random order, which is different for everyone as well. User feedback is needed to decide whether this is acceptable in all situations (it appears to be, so far), or whether we should simulate a ‘round table’. Some of the ideas of the WITlab group at TU Delft [ref?] could be taken into account here.

Meeting currently takes an anarchist standpoint: whoever wants to speak can do so. It might be a good idea, especially for bigger meetings, to add a ‘permission-to-talk’ system such as is used in real meeting rooms. Whenever you have something to say you request permission by clicking your talk indicator, and the chairman grants permission (or takes it away again) by also clicking your indicator light. A four-colour indicator would convey the state information (quiet, quiet but asking for permission to speak, quiet with permission to speak and speaking) to the users. There seems to be work done in the area of systems ensuring a fair share in speech time to all participants, but this looks like overkill (from my experiences with meetings, at least).

Some other possible extensions are the possibility to talk to people in private (maybe even extended to a concept of talking to a subset of the people in the meeting) and a system to invite people to join a meeting. If a reasonable paradigm can be devised this might even lead to a system resembling Internet Relay Chat, but voice based: lots of people have a tiny chat window and are subscribed to numerous interest groups. As soon as someone has to say something it gets transmitted to all the people subscribed to the relevant group. This could lead to all sorts of interesting usage, like peer-help with computer problems (remember the terminal rooms of the seventies, where you would first ask the other people around for a solution before digging up a manual?) and other things. This would, however, require the subscription protocol to be reverted to a UDP based distributed protocol in stead of the current centralised TCP based protocol. The protocol used in vat [ref] might provide some clues.

Finally, a connection to the real phone system would be nice. The only thing needed for this would be one or more daemons somewhere on the network that are connected to the normal phone system. Combined with a database, meeting could show external peoples’ faces, dial the correct number, presenting a recording to the answering party asking whether or not they want to be in the meeting, etc. The hardware has been built, but the software is not in place yet.